Lecture 13: ISA and Assembly

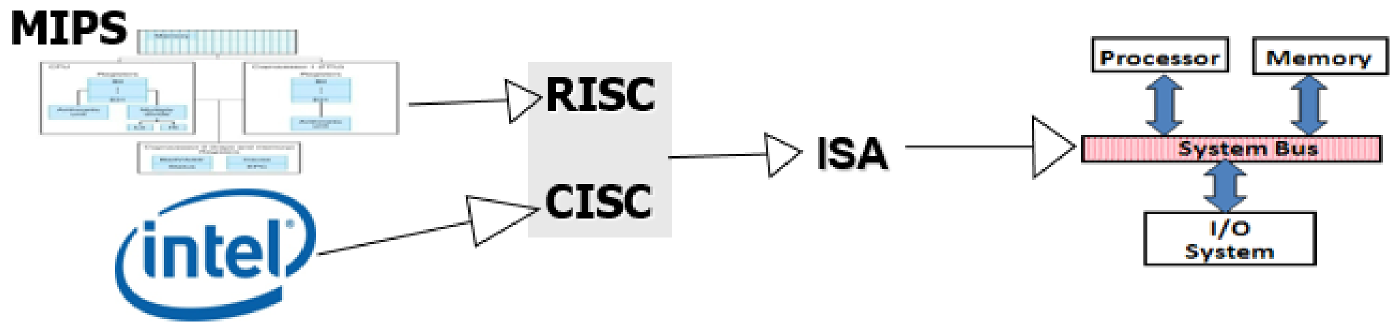
Topics

- More on ISA
  - RISC vs. CISC
  - **MIPS** architecture summary
  - **Intel** architecture
- Interpret binary patterns
- Final examination

> You should have learned a lot from this unit, not only the technical part, but also logical thinking.
> Understood the low level mechanics of computer systems. Gained an insight into SPIM and Computer Organisation.
> Laboratories were very helpful and effective in assisting learning.
> Assembly programming reinforces understanding of high-level programming constructs.
> The unit is **difficult** to study and the study load is heavy. You've learnt how to actively work under a pressure in workplaces.
> Much effort has been put on computer organisation and design.

The Architecture family

- Architecture Modeling at different levels



	RISC	CISC
Implementation	RISC directly in hardware	CISC microprogrammed
Instruction sets	Instructions are simple; Load / Store architecture	Trend towards instruction set similar to high-level languages
Speed of execution	faster in RISC (simpler hardware)	
Size of executable file		smaller in CISC (less memory used)

Reduced Instruction Set Computer (RISC)

- Characteristics
  - fixed size of instructions
  - few instruction formats
  - load/store (or: register-register) – moves data or operates on data, NOT at the same time
  - few addressing modes
  - large number of general registers (operands are always in registers)
  - few (if any) hidden register usage
- Reduced
  - not necessarily a small number of instruction
  - but definitely simpler instructions

Complex Instruction Set Computer (CISC)

- Characteristics
  - variable size instructions
  - many instruction formats (varied no of arguments)
  - many addressing formats
  - small number of general registers (more related to technology)
  - implied usage of special registers
    - for example condition codes
- Complex
  - a single instruction may execute a complex function
    - for example a value of a polynomial function
  - there is a large number of instructions

Intel history: ISA evolved since around 1970

- 4004 (released 1971)
  - 4-bit, US$299 each, 108Khz (developed for calculators, also used in various controllers)
- 8008 (1972)
  - 200Khz, Radio Electronics magazine describes Mark-8, 1st home computer
- 8086 (1978)
  - 16-bit, all internal registers 16 bits; no general purpose registers, 5-10MHz
- 8087 (1980)
  - adds 60 FP instructions, adds 80-bit-wide stack, but no registers
- 80286 (1982)
  - adds elaborate protection model, address extended to 24 bits, 6-12.5MHz
- 80386 (1985)
  - 32-bit; converts 8 16-bit registers into 8 32-bit general purpose registers; new addressing modes; adds paging, 16-33MHz

Intel history cont.

- 80486(1989)
  - + 17 new instructions, FP unit on the same chip, 25-50MHz
- Pentium, Pentium MMX (1993)
  - MMX adds 57 instructions for multimedia, up to 233MHz
- P6 family (Pro, II, III) (1995)
  - +70 instructions for multimedia, multiprocessing, up to 1.2GHz
- Pentium 4 (2000)
  - +76 SIMD (former MMX) instructions, additional FP unit. May 2003: up to 3.06GHz. Price for 2.8GHz approx. the same as 4004 in 1971.
  - 2003: Pentium 4 with Hyper-Threading technology
- Centrino mobile technology (2003)
  - three components work together: low power consumption Pentium M CPU, wireless LAN controller, and chipset
- Itanium, Itanium 2 processors family (current)
  - **64-bit** computing platform, back compatible with 32-bit software
- Dual core models, multiple core models … (current)

MIPS vs. 80386 (32-bit)

Note that both also offer 64-bit architectures, not covered here.

- MIPS (RISC)
  - Address: 32-bit
  - Instruction size 32 bits
  - Data aligned
  - Destination reg: Left

    add **$rd**,$rs1,$rs2

    # $rd=$rs1+$rs2
  - Regs: $0, $1, ..., $31
  - Reg = 0: $0
  - Return address: $31

- Intel 80386 (CISC)
  - 32-bit
  - Instruction size 1-17 bytes
  - Data unaligned
  - Right*

    add %rs1,**%rs2**

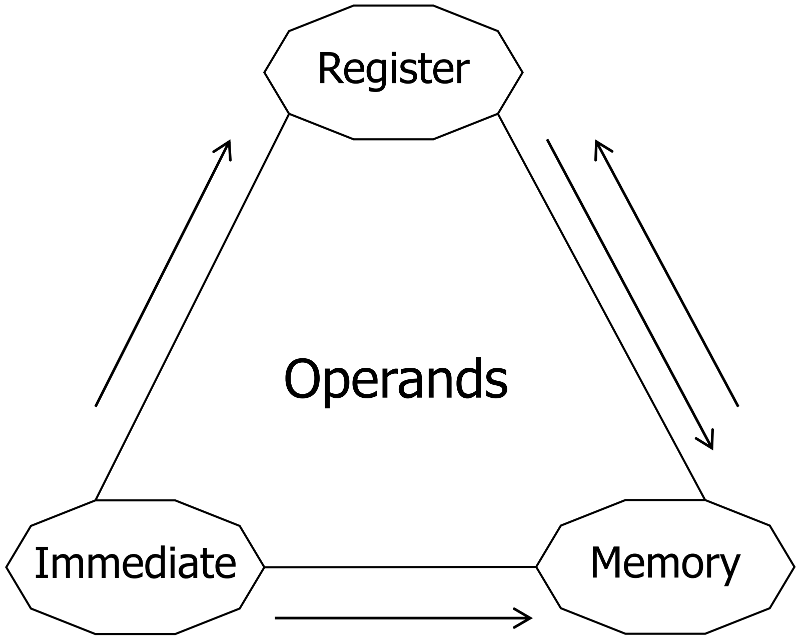
    # %rs2=%rs1+%rs2
  - %r0 (%eax), %r1, ..., %r7
  - all have some special purpose
  - implicit use in instructions

*The GNU Assembler, gas, uses a different syntax from what you will likely find in any x86 reference manual, and the two-operand instructions have the source and destinations in the opposite order.

MIPS vs. Intel 80x86

- Instruction examples

MIPS	Intel 80x86
addu, addiu	add**l**
subu	subl
and,or, xor	andl, orl, xorl
sll, srl, sra	sall, shrl, sarl
lw	movl mem, reg
sw	movl reg, mem
mov	movl reg, reg
li	movl imm, reg
lui	n.a.



MIPS vs. Intel 80x86

- MIPS (also Alpha): "**fixed-length** instructions"
  - All instructions same size, e.g., 4 bytes
  - simple hardware ⇒ performance
  - branches can be multiples of 4 bytes
- X86 (also VAX): "**variable-length** instructions"
  - Instructions are multiple of bytes
  - small code size (30% smaller?)
  - More Recent Performance Benefit: better instruction cache hit rates
  - Instructions can include 8- or 32-bit immediates

MIPS vs. Intel 80x86

- MIPS: “Three-operand architecture”
  - Arithmetic-logic specify all 3 operands

    add $s0,$s1,$s2 # s0=s1+s2
  - Benefit: fewer instructions ⇒ performance
- x86: “Two-operand architecture”
  - Only 2 operands,
  - so the destination is also one of the sources, for example:

    add $s1,$s0 # s0=s0+s1; s0+=s1
  - Often true in C statements: c += b;
  - Benefit: smaller instructions ⇒ smaller code

MIPS vs. Intel 80x86

- MIPS: "load-store architecture"
  - Only Load/Store access memory; rest operations register-register; e.g.,

    ```
    lw $t0, 12($gp)
    add $s0,$s0,$t0 # s0=s0+Mem[12+gp]
    ```
  - Benefit: simpler hardware ⇒ performance
- X86: "register-memory architecture"
  - All operations can have one operand in memory; other operand is a register; e.g.,

    ```
    add 12(%gp),%s0 # s0=s0+Mem[12+gp]
    ```
  - Benefit: fewer instructions ⇒ smaller code

80386 addressing

- General format

  base reg + index * scale + displacement

  [displacement from 1 to 4 bytes long]

- Examples:
  - base reg + offset (like MIPS)

    movl -8000044(%ebp),%eax # eax = Mem[ebp - 8000044]
  - base reg + index_scaled reg (like VAX)

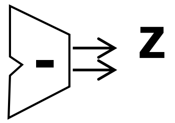
    movl (%eax,%ebx, 4),%edi   # edi = Mem[ebx*4 + eax]
  - base reg + index_scaled reg + offset

    movl 12(%eax,%edx,4),%ebx # ebx = Mem[edx*4 + eax + 12]

Branch in 80x86

- Rather than compare registers, x86 uses special **1-bit registers** called "condition codes" [flag] that are set as a side-effect of ALU operations
  - S - Sign Bit
  - Z - Zero (if result is all 0)
  - C - Carry Out
  - P - Parity: set to 1 if there are even number of ones in rightmost 8 bits of operation
- Conditional Branch instructions then use condition flags for all comparisons: <, <=, >, >=, ==, !=

S=0 **JNS** Jump if positive number i.e. if sign bit is clear
S=1 **JS** Jump if negative number i.e. if sign bit is set
Z=0 **JNE** Jump if a-b not equal to 0 i.e. if Zero bit clear (Zero **!=** 1)
Z=1 **JE** Jump if a-b equal to 0 i.e. if Zero bit is set (Zero **==** True)
C=0 **JNC** Jump if carry bit is clear (Jump if no carry)
C=1 **JC** Jump if carry bit is set



```
    cmpl %ebx, %ecx
    jne SkipStmts
    incl %eax
SkipStmts:
```

Unusual features of 80x86

- Terminology
  - 16-bits called word (halfword in MIPS)
  - 32-bits double word or **l**ong word (word in MIPS)
- 8 32-bit Registers have names - with "**e**" prefix (for Extended):
  - e**a**x (r0), e**b**x, e**c**x, e**d**x, esp, ebp, esi, edi
- PC is called **eip** (instruction pointer)
- leal (load effective address - 32-bit)
  - Calculate address like a load, but load address into register, not data
  - Load 32-bit address:

    leal -4000000(%ebp),%esi # esi = ebp – 4000000
- Positive constants start with **$**; regs with **%**
  - cmpl $999999,%edx

Unusual features of 80x86

- ***loop*** is an instruction for programming loops
  - it is a conditional jump instruction
  - it uses the **%ecx** register as a count
  - decrements %ecx, and jumps if not zero
- Memory Stack is part of instruction set
  - ***call*** places return address onto stack, increments **esp** (Mem[esp]=eip+6; esp+=4)
  - ***push*** places value onto stack, increments esp
  - ***pop*** gets value from stack, decrements esp
- ***inc[l], dec[l]*** (increment, decrement)
  - incl %edx   #edx = edx + 1
  - Benefit: smaller instructions ⇒ smaller code

Unusual features of 80x86

- Floating point uses a separate stack (fpstack*)

  load/push operands; perform operation (e.g. sub); store/pop result

```
flds -8000048(%ebp)
        # push M[ebp-8000048]
fildl (%esp)
        # fpstack = M[esp],
        # convert integer to FP
fsubp %st,%st(1)
        # subtract top 2 elements
fstps -8000048(%ebp)
        # M[ebp-8000048] := difference
```

*Intel has actually created three separate generations of floating-point hardware for the x86.
1) they started with a **stack-oriented FPU** modeled after a pocket calculator in the early 1980's;
2) started over again with a register-based version called SSE in the 1990's; and
3) have just recently created a three-input extension of SSE called AVX.

And in Conclusion ...

- Once you've learned one RISC instruction set, easy to pick up the rest
  - ARM, Compaq/Digital Alpha, Hitachi SuperH, HP PA, IBM/Motorola PowerPC (Apple Mac), Sun SPARC, ...
- Intel 80x86 is a different thing all together ...
- RISC emphasis: performance, HW simplicity
- CISC emphasis: code size
- Intel x86:
  - very long lived
  - many additions not necessarily well fitting with the original design
- BUT: distinction between "RISC" and "CISC" is often blurred, both architectures often use similar solutions

Interpretation of binary patterns

0x34554342

0	0	1	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0

- What does this bit pattern mean:
  - 878,003,010?
  - “4UCB”?
  - $2.034*10^{-4}$?
  - ori $s5, $v0, 17218?
- If a memory location contains this pattern, can its meaning be determined?
  - In other words: What is the type of data?

- Operation on instruction that accesses operand determines its type!
  - Side-effect of stored program concept: instructions stored as numbers
- Power/danger of unrestricted addresses/pointers:
  - Use ASCII as FP, instructions as data, integers as instructions, ... (Leads to security holes in programs, errors, etc.)

Data types in MIPS

- The interpretation depends on the context
  - unsigned binary number
  - signed binary number
  - floating point number
  - a half of a double precision FP number
  - collection of 4 bytes
  - collection of 2 halfwords
  - collection of 32 bits
  - an instruction

- An unsigned integer
  - **Addu**
- A signed integer
  - **add**

- FP number
  - **add.s**
- FP double precision
  - **add.d**

- An 8 bit field
  - lb, sb
- A 16 bit field
  - lh, sh
- 32 bits
  - xor

When is it an integer number

- If it is used in integer calculations
  - we are talking about values here not representation
- the integer numbers in MIPS are always represented as binary numbers
  - MIPS has no instructions to handle integer values represented in any other way

When is it a string of characters

- From "raw" hardware point of view NEVER
  - there are no instructions in the MIPS instruction set which are specifically provided for handling **characters**
  - there are instructions which can handle **single bytes**
  - a single byte may be an ASCII character, but it may also be an EBCID character (IBM standard), a pixel or anything else which fits into 8 bits
- From the OS point of view
  - there are OS services specifically provided for handling characters and character strings
  - a single byte is a character if it is handled as a character
- From an application program point of view
  - other interpretations are possible

When is it an instruction

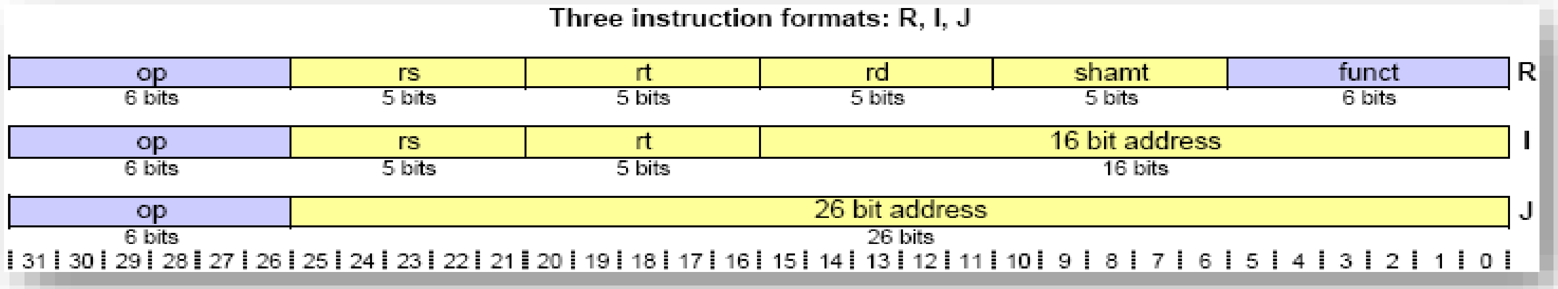
- From “raw” hardware point of view
  - if its **address** is loaded into **PC**
- With memory allocation conventions
  - if it is in the **text segment** of memory
- MIPS only knows it is an instruction if it is used as an instruction
  - there is **nothing inherent** in the bit pattern which makes it an instruction, or anything else

Encode instruction

Name	ID	Value
$zero	$0	**0 unchangeable**
$t0	$8	can be **8**; but changeable

- Three instruction format: R, I, J

see HP4, P134; *instruction decoding.pdf*



Three instruction formats: R, I, J
op 6 bits | rs 5 bits | rt 5 bits | rd 5 bits | shamt 5 bits | funct 6 bits | R
op 6 bits | rs 5 bits | rt 5 bits | 16 bit address 16 bits | I
op 6 bits | 26 bit address 26 bits | J
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

- Instruction assembling that converts mnemonic format to machine code

```
                         rd     rs     rt    ①
R-type instruction:  add $t0,   $s1,   $s2          [mnemonic]
                     add  8     17     18    ②      [assembled]
```

③

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	op	**rs**	**rt**	**rd**	**shamt**	**funct**
④ dec	0	17	18	8	0	32
⑤ bin	0 0 0 0 0 0	1 0 0 0 1	1 0 0 1 0	0 1 0 0 0	0 0 0 0 0	1 0 0 0 0 0

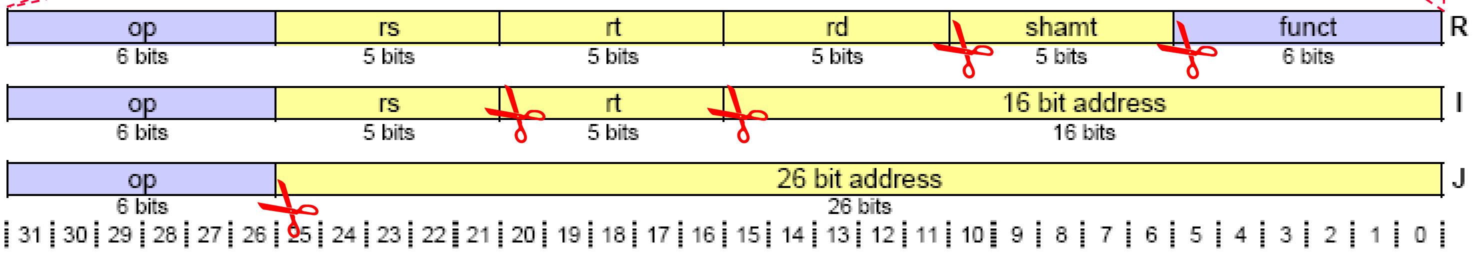
⑥ 0x	**0**	**2**	**3**	**2**	**4**	**0**	**2**	**0**

0x02324020 0x34020005

Decode instructions

see HP4, P134; *instruction decoding.pdf*

Three instruction formats: R, I, J



op 6 bits | rs 5 bits | rt 5 bits | rd 5 bits | shamt 5 bits | funct 6 bits | R
op 6 bits | rs 5 bits | rt 5 bits | 16 bit address 16 bits | I
op 6 bits | 26 bit address 26 bits | J
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

NOTES:

See the lectures 2, 3 and the textbook Patterson and Hennessey "Computer Organization & Design" Ch3 (Ed2) or Ch2 (Ed3).

1 word -- 32 bits

Two examples of translating a machine instruction into a MIPS assembly instruction

0x 02324020 - what MIPS instruction is it?

0				2				3				2				4				0				2				0			
0	0	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

op	rs	rt	rd	shamt	funct
0 0 0 0 0 0	1 0 0 0 1	1 0 0 1 0	0 1 0 0 0	0 0 0 0 0	1 0 0 0 0 0
	16 1	16 2	8		dec: 32 or hex: 20
R-type instruction	R17=$s1	R18=$s2	R8=$t0	not used by "add"	add

ANSWER: add $8, $17, $18 ==> add $t0, $s1, $s2 add rd, rs, rt

op (operation) field tells us that this is R-type instruction (see the textbook from p.117, Fig. 3.18 or Fig. A.19). R-type instructions fields are allocated in groups: 6bits-5bits-5bits-5bits-5bits-6bits. Funct field tells us that the instruction is 'add', format: add rd, rs, rt (p. A-55).

A_{50}

0x 34020005 - what MIPS instruction is it?

3				4				0				2				0				0				0				5			
0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

op	rs	rt	imm
0 0 1 1 0 1	0 0 0 0 0	0 0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
dec: 13 or hex: 0d		2	4 1
ori	R0=$r0	R2=$v0	5

ori rt, rs, imm ANSWER: ori rt, rs, imm ==> ori $2, $0, 5 ==> ori $v0, $0, 5

op field tells us that this instruction is 'ori' (see the textbook from p.117, Fig. 3.18 or Fig. A.19). ori is I-type instructions, so fields are allocated in groups: 6bits-5bits-5bits-16bits. The instruction format is: ori rt, rs, imm (p. A-57).

© Derek Bem, 2005

Assembly language vs HLL

- Assembly language
  - very simple instructions
  - one instruction per line
  - programs difficult to understand
  - registers and memory locations
  - cannot enforce data types, depends on instruction
  - total control of register usage
  - unrestricted use of instructions (hardware specific features)
- High level language
  - statements
  - statements expand to many instructions
  - programs easier to understand
  - simple variables, arrays, structures
  - can enforce data types (if no pointers)
  - register usage determined by the compiler
  - only what is allowed to compiler

Big issues - cost / performance

- Programming in assembly language takes more time
  - productivity (constant no of lines per day)
  - more errors (programs longer, more cryptic, no type enforcement)
- Maintaining assembly language program is harder
  - same reasons as above
  - few skilled programmers
- Higher run-time performance
  - was true before optimising compilers available
  - still possible for short sections
  - requires great skill
  - improvements in algorithm more important

So why use it at all?

- Embedded systems
- If no compiler is available
- Operating systems
  - require direct access to all available resources (physical memory and I/O devices, privileged registers and instructions)
  - commonly only small parts of OS in assembly language
- Maintenance of existing programs
  - so called legacy software
  - may be preferable to rewriting in a high level language
- … …

FINAL EXAM

The Thinker
Rodin 1840-1917

- OPEN BOOK exam; Max. total mark is 50 (and counts for 50% of total mark)
  - All printed materials, books, handwritten or printed notes are allowed. Laptops and Calculators are **NOT** allowed.
- Similar in nature to written submissions for the workshops and exercises from lectures
  - **Requires understanding of concepts, not memorizing facts**
  - Each question may have a number of sub-questions
- May require simple arithmetic
  - but: arithmetic mistakes are not deadly
  - it is more important to show how you arrived at the numerical answer, then to give final numerical answer.

DO

- BEFORE the exam: try to solve sample exam questions placed on vUWS; exam questions are always somewhat similar.
- During final exam, attempt to answer all questions
- Answer questions from the easiest to the hardest
  - the exam questions are not printed in any particular order
  - number your answers (e.g. Q1.(b).ii.), so I know which question you are answering
- Bring only printed materials you are familiar with
  - your own notes, and annotated lecture notes
  - your own written workshop submissions
  - the textbook you used for studying
  - no calculators are needed, laptops (with or without wireless LAN) are NOT allowed, miniature radio transmitters/receivers are also not allowed.

DON'T

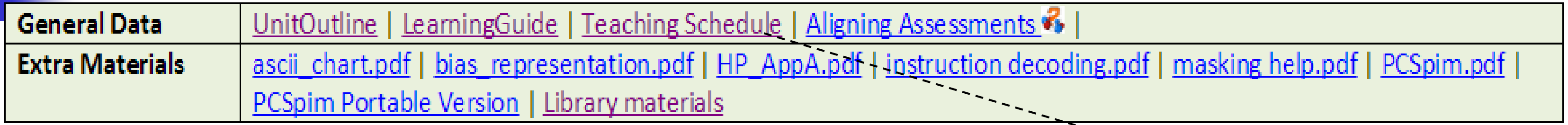
- Supervisors running the exam contact me only if there is an unexpected situation, for example: obvious misprint in the exam paper, a page is missing, a page is unreadable because of photocopying error, etc. However they will NOT contact me with queries along the lines: "how do I start answering this one?", "what this question means?", "what is the meaning of life?" etc.
- DON'T fragment the answers to a single question
  - I may not find all bits and pieces in your paper if they are written on separate pages
- DON'T answer the questions in the paper order
  - this order is NOT "from the easiest to the hardest"!
  - Answering order: Q. 1, 2, 3… etc. is most likely not the best for you
- DON'T spend time on hard questions
  - you may get bogged down on a single question
  - and run out of time for other questions which could earn you marks
- DON'T copy parts from your materials hoping that it 'may be close' and you have nothing to lose (this approach = mark 0)

Deferred (Supplementary) Exam

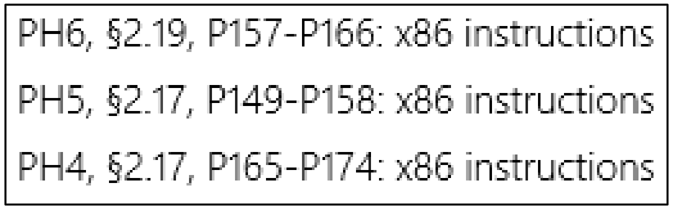
- Only for students who had a genuine reason not to sit the final exam (administration decides that, I do not)
  - If you feel sick on the day (or have urgent matter/misadventure that will really affect you in the exam), **don't sit/attempt** the exam. Get a doctor's certificate (or consult student central) for a deferred exam.
  - Special consideration will **NOT** be given to students who sit the exam and then go to the doctor afterwards.
  - It does not count if you suddenly start feeling sick after reading the exam paper, because you realised you can't answer any question.
  - In the past deferred/supplementary was not harder than the final, but:
    - the failure rate was several times that of the final…
  - The University admin is very strict with the deferred exam permissions (again: it is not my decision who should be allowed to do the supplementary exam).

GOOD LUCK!!!

Recommended readings



General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials



PH6, §2.19, P157-P166: x86 instructions

PH5, §2.17, P149-P158: x86 instructions

PH4, §2.17, P165-P174: x86 instructions

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)

https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263